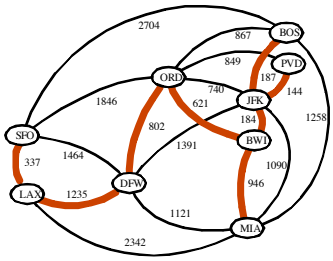
Arbre couvrant minimal
(Minimal spanning tree)



Arbre couvrant minimal

- Définitions
- L'algorithme de Prim-Jarnik
- L'algorithme de Kruskal

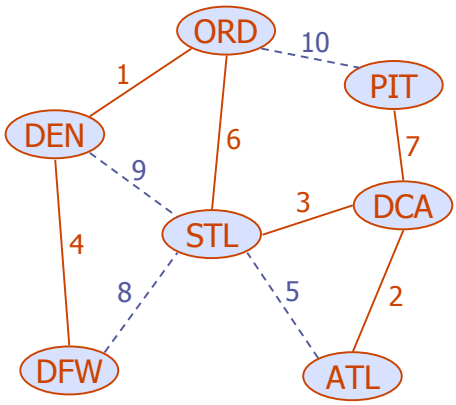
Arbre couvrant minimal

Sous-graphe couvrant
- Sous-graphe d'un graphe G contenant tous les sommets de G

Arbre couvrant
- Sous-graphe couvrant qui est un arbre

Arbre couvrant minimal (*ACM*)
- Arbre couvrant d'un graphe pondéré avec la somme des poids des arêtes qui constituent l'arbre est minimal

- Applications
  - Réseaux informatiques
  - Réseaux routiers

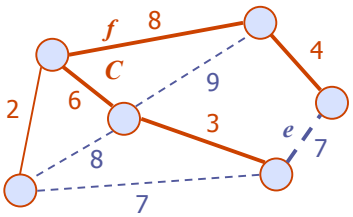


Propriété de cycles

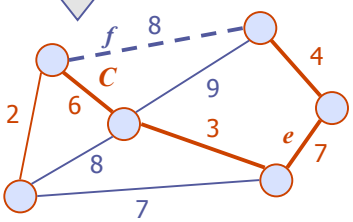
Propriété de cycles:
- Soit T un arbre couvrant minimal d'un graphe pondéré G
- Soit e une arête de G n'appartenant pas à T et soit C le cycle obtenu lorsqu'on ajoute e à T
- Pour chaque arête f dans C, $poids(f) \leq poids(e)$

Preuve:

Par contradiction
- Si $poids(f) > poids(e)$ nous pouvons obtenir un arbre couvrant de plus petit poids en remplaçant e par f

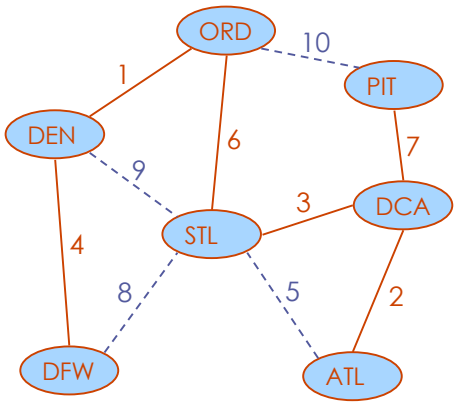


Remplaçant f avec e produit un mieux arbre couvrant



Propriété de cycles

Dans tout cycle du graphe, les arêtes faisant partie d'un ACM ont des poids inferieurs aux autres arêtes (en pointillé)



5

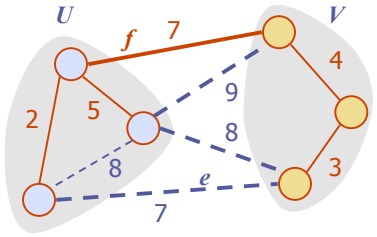
Propriété de partition

Propriété de partition:

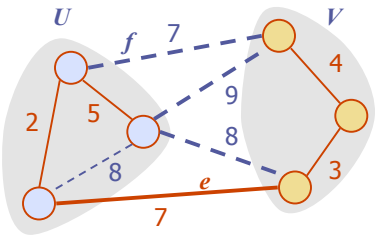
Considérons une partition des sommets de G en deux ensembles U et V. Soit e une arête de poids minimal entre U et V. Alors, il existe un arbre couvrant minimal de G contenant e

Preuve:

- Soit T un arbre couvrant minimal de G
- Si T ne contient pas e, soit C le cycle formé par l'addition de e à T et soit f une arête de C entre U et V
- Par la propriété de cycles,
  $poids(f) \leq poids(e)$
- Alors, $poids(f) = poids(e)$
- Nous obtenons un autre ACM en remplaçant f par e



Remplacer **f** avec **e** donne un autre ACM



6

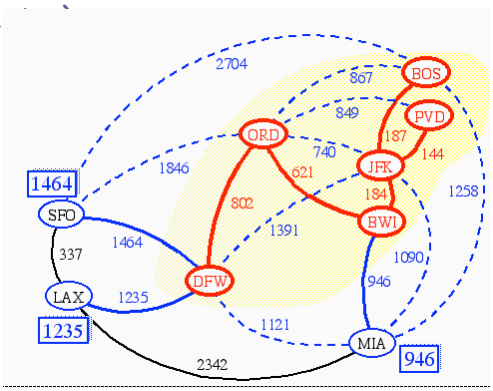
L'algorithme de Prim-Jarnik

- L'algorithme de Prim-Jarnik pour calculer un ACM est similaire à l'algorithme de Dijkstra
- Nous supposons que le graphe est connexe
- Nous choisissons un sommet arbitraire s et nous faisons grossir l'ACM comme un nuage de sommets, commençant de s
- Nous emmagasinons avec chaque sommet v une étiquette $d(v)$ représentant le plus petit poids d'une arête connectant v à n'importe quel sommet dans le nuage (comme opposé à la somme totale des poids sur un chemin du sommet de départ jusqu'à u).

7

L'algorithme de Prim-Jarnik

- Pour chaque étape
  - Nous ajoutons au nuage le sommet extérieur u ayant la plus petite étiquette de distance
  - Nous mettons à jour les étiquettes des sommets adjace



- Utiliser une file à priorité Q dont les clés sont les étiquettes D, et dont les éléments sont des paires sommet-arête.
  - Clé : distance
  - Élément: sommet
- Tout sommet *v* peut être le sommet de départ.
- On continue à initialiser toutes les valeurs de D[*u*] à "infini", mais nous initialisons aussi E[*u*] (l'arête associé à *u*) à "null".
- Retourne l'arbre recouvrant minimal T.
- *Nous pouvons réutiliser le code produit par Dijkstra, et ne changer que quelques parties. Observons le pseudo-code....*

9

Algorithm PrimJarnik(G):

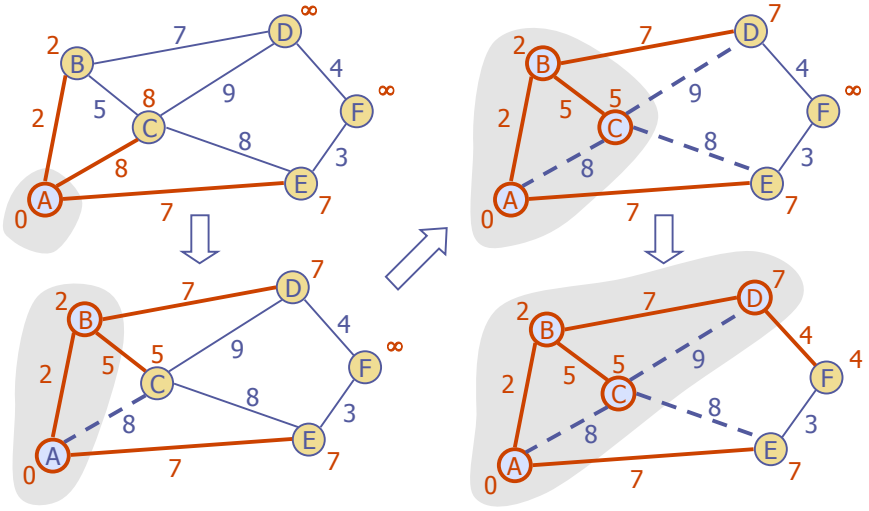
Entrée: Un graphe pondéré *G*.

Sortie: Un ACM *T* pour *G*.

```
pick any vertex v of G
{grow the tree starting with vertex v}
 T ← {v}
 D[u] ← 0
 E[u] ← ∅
 for each vertex u ≠ v do
   D[u] ← ∞
 let Q be a priority queue that contains vertices, using the D labels as keys
 while Q ≠ ∅ do  {pull u into the cloud C}
    u ← Q.removeMinElement()
    add vertex u and edge E[u] to T
   for each vertex z adjacent to u do  if z is in Q
    {perform the relaxation operation on edge (u, z) }
         if weight(u, z) < D[z] then
           D[z] ←weight(u, z)
           E[z] ← (u, z) change the key of z in Q to D[z]
        return tree T
```

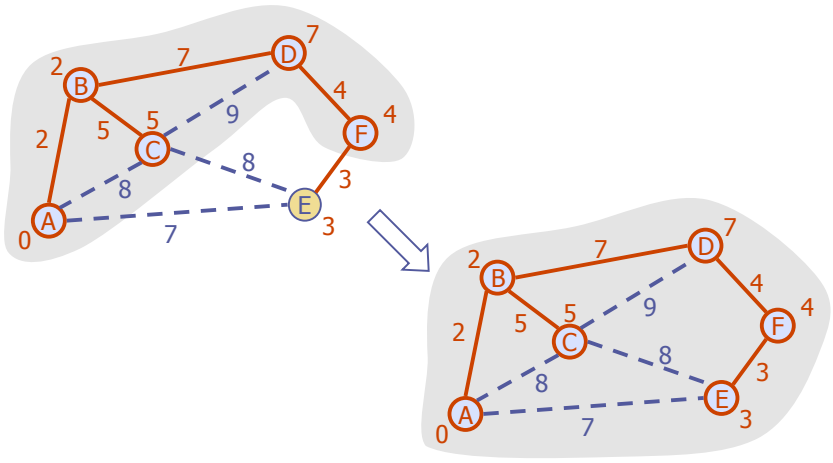
10

Exemple



11

Exemple (suite)



12

Prim-Jarnik…
Pourquoi il marche

- Ceci est une application de la propriété de Cycles!
- Soit (u,v) l'arête minimale dans quelques itérations. S'il y a un ACM ne contenant pas (u,v), alors (u,v) complète un cycle. Puisque (u,v) a été considéré avant quelque autre arête connectant v au groupe, il doit avoir un poids inférieur ou égal au poids de l'autre arête. Un nouveau ACM peut être formé par permutation

Analyse de complexité

- Opérations sur les graphes
  - Méthode incidentEdges est appelée une fois pour chaque sommet
- Opérations d'étiquetage
  - Nous plaçons/obtenons les étiquettes de sommet z $O(\deg(z))$ fois
  - placer/obtenir les étiquettes prend $O(1)$
- Opérations de file de priorité
  - Chaque sommet est inséré une fois et enlevé une fois de la file à priorité, où chaque insertion ou suppression prend $O(\log n)$
  - La clé d'un sommet w dans la file à priorité est modifiée au plus $\deg(w)$ fois, où chaque changement de clé prend $O(\log n)$

- L'algorithme de Prim-Jarnik est exécuté en $O((n + m) \log n)$ si le graphe est représenté par la structure de liste d'adjacence
  - Rappelez-vous que $\sum_v \deg(v) = 2m$
- Le temps d'exécution est $O(m \log n)$ parce que le graphe est connexe

Dijkstra vs. Prim-Jarnik

```
Algorithm DijkstraShortestPaths(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based priority queue
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

L'algorithme de Kruskal

- Chaque sommet est emmagasiné initialement comme son propre groupe.
- A chaque itération, l'arête de poids minimal dans le graphe est ajoutée à l'arbre couvrant si elle relie 2 groupes distincts.
- L'algorithme se termine quand tous les sommets sont dans le même groupe.
- Ceci est une application de la propriété de partition!
- Si l'arête minimale dans quelques itérations est (u,v), alors si nous considérons une partition de G avec u dans un groupe et v dans l'autre, alors la propriété de partition dit qu'il doit y avoir un ACM contenant (u,v)

17

L'algorithme de Kruskal

- Une file de priorité emmagasine les arêtes extérieures (hors du nuage)
  - clé: poids
  - élément: arête
- À la fin de l'algorithme
  - On se retrouve avec un nuage qui entoure l'ACM
  - Un arbre *T* qui est notre ACM

```
Algorithm KruskalMST(G)
  for each vertex V in G do
    define a Cloud(v) of ← {v}
  let Q be a priority queue.
  Insert all edges into Q using their
weights as the key
  T ← ∅
  while T has fewer than n-1 edges do
    edge e = T.removeMin()
    Let u, v be the endpoints of e
    if Cloud(v) ≠ Cloud(u) then
      Add edge e to T
      Merge Cloud(v) and Cloud(u)
  return T
```

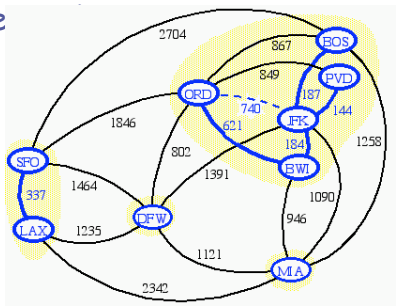
18

Structure de données pour l'algorithme de Kruskal

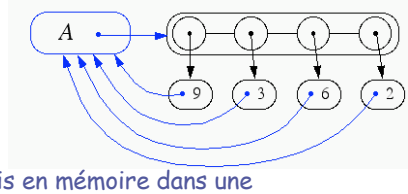
- L'algorithme maintient une forêt d'arbres
- Une arête est acceptée, si elle relie deux arbres distincts
- Nous avons besoin d'une structure de données qui maintient une partition c.-à-d. une collection d'ensembles disjoints, avec le

  -**find**(u): retourne l'ensemble contenant u

  -**union**(u,v): remplace les ensembles contenant u et v par leur union



Représentation d'une partition:



- Chaque élément d'un ensemble est mis en mémoire dans une séquence (l'ensemble pointe vers la séquence contenant ces éléments)
- Chaque élément à un pointeur vers l'ensemble
  - L'opération find(u) se fait en $O(1)$ et retourne l'ensemble dont u fait partie
  - Pour l'opération union(u,v), nous bougons les éléments du plus petit ensemble dans la séquence du plus grand ensemble et nous mettons à jour leur pointeur
  - La complexité en temps de union(u,v) est $\min(n_u,n_v)$, où n_u et n_v sont les tailles des ensembles contenant u et v
- Chaque fois qu'un élément est traité, il est mis dans un ensemble de taille au moins double, donc chaque élément est traité au plus log n fois

Implémentation à base de partition

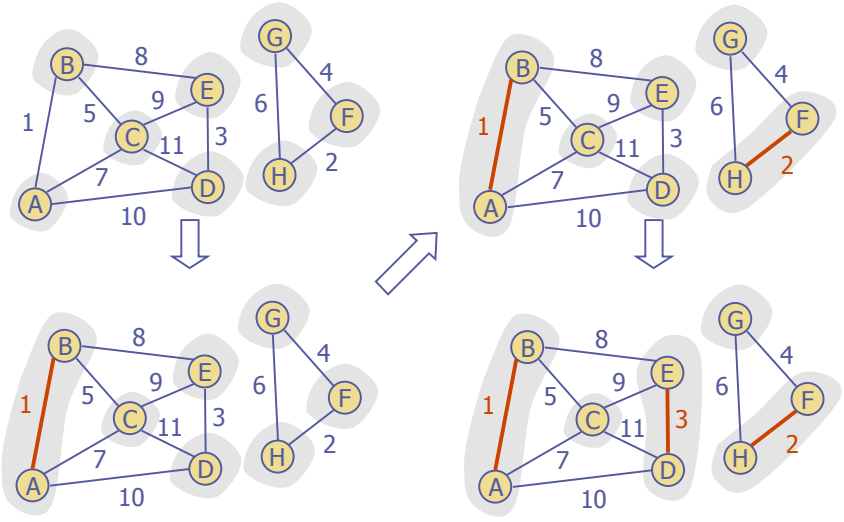
- Une version à base de partition de l'algorithme de Kruskal exécute les fusions du nuage comme des union(u,v) et les tests comme des find(u).

Algorithm Kruskal(*G*):
  Entrée: Un graphe pondéré *G*.
  Sortie: Un ACM *T* pour *G*.
Soit *P* une partition des sommets de *G*, où chaque sommet est dans un ensemble séparé.
Soit *Q* une file à priorité gardant en mémoire les arêtes de *G*, ordonnées selon leur poids
Soit *T* un arbre initialement vide
while *Q* n'est pas vide **do**
  (*u*,*v*) ← *Q*.removeMinElement()
  if *P*.find(*u*) != *P*.find(*v*) **then**
    Ajouter (*u*,*v*) à *T*
    P.union(*u*,*v*)
return *T*

Temps: O((n+m)log n)

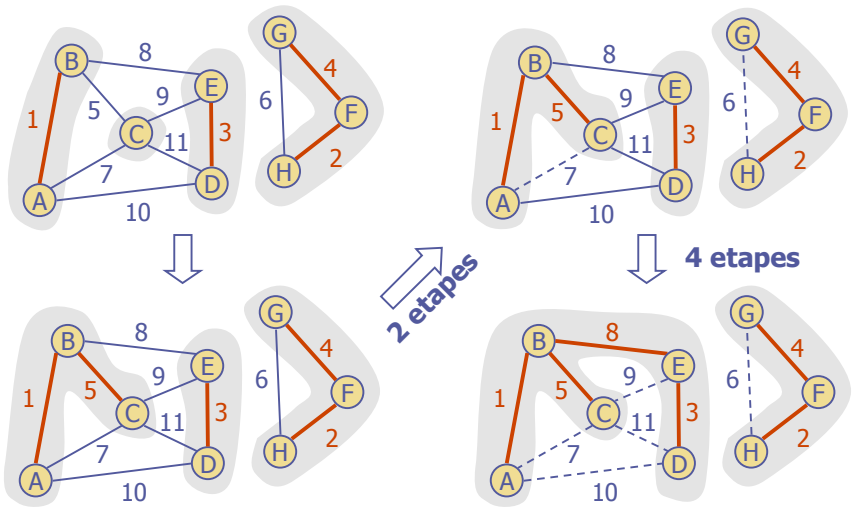
21

Exemple



22

Exemple (suite)



23